The SICStus Emulator

by

Mats Carlsson

September 1991

Swedish Institute of Computer Science
Box 1263, S-164 29 KISTA, SWEDEN
The SICStus Emulator

Mats Carlsson

SICS technical report T91:15
Swedish Institute of Computer Science
Box 1263, S-164 28, KISTA, Sweden

September 2, 1991

Abstract

This report documents internal details of Industrial SICStus Prolog, developed at the Logic Programming Laboratory of SICS. The development was funded by Ericsson Telecom AB, NobelTech Systems AB, Infologics AB and Televerket under the IT4-program. The implementation was done on Sun-3 and Sun-4 workstations. It has proved portable to a wide range of byte-addressed 32-bit computers running Berkeley Unix.

Contents

1 Introduction

2 Language and Implementation
   2.1 The Language .................................................. 1
   2.2 The Implementation .......................................... 1

3 Storage Model
   3.1 Terms and their Representation ............................. 2
   3.2 Data Areas .................................................... 4
      3.2.1 The Static Area ........................................... 4
      3.2.2 The Heap .................................................. 5
      3.2.3 The Environment Stack ................................. 5
      3.2.4 The Chreopoint Stack .............................. 6
      3.2.5 The Trail Stack ........................................ 6
   3.3 Abstract Machine Registers ................................. 7

4 The Emulator
   4.1 Interfaces .................................................... 7
   4.2 Instruction Set ............................................... 9
      4.2.1 Put Instructions ......................................... 9
      4.2.2 Get Instructions ....................................... 11
      4.2.3 Unify Instructions ..................................... 13
      4.2.4 Procedural Instructions .............................. 16
      4.2.5 Indexing Instructions .............................. 17
      4.2.6 Utility Instructions .................................. 19
   4.3 General Unification ......................................... 22
5 Builtin Predicates and Functions  24
  5.1 Inline Predicates and Functions ......................... 24
  5.2 Predicates Called From C ................................ 25
  5.3 Builtin C Predicates ..................................... 25

6 Qload Instruction Set  39

7 The Garbage Collector  42
  7.1 Variable Shunting ........................................ 42
  7.2 Support for Constrained Variables ...................... 42
  7.3 Support for Big Integers and Floats .................... 42

8 The Compiler  43
  8.1 Support for Big Integers and Floats .................... 43
  8.2 Support for Native Code Back-Ends ...................... 43
  8.3 Support for Profiling .................................. 43
  8.4 Support for Modules .................................. 44

A Emulator Macros  45

B Bytecode Representation  48
  B.1 Instruction Group 1 .................................... 48
  B.2 Instruction Group 2 .................................... 59

References  61
1 Introduction

This report documents internal details of Industrial SICStus Prolog, developed at the Logic Programming Laboratory of SICS. The development was funded by Ericsson Telecom AB, NobelTech Systems AB, Infologics AB and Televerket under the IT4-program. The implementation was done on Sun-3 and Sun-4 workstations. It has proved portable to a wide range of byte-addressed 32-bit computers running Berkeley Unix.

The semantics of the various instructions will be given in a style similar to the C programming language. The descriptions of the instructions will closely follow the actual emulator code. However, a number of optimizations have been omitted from this exposition in order to (we hope) make it clearer.

The report is organized as follows: Section 2 outlines the implementation, Section 3 defines the storage model of the abstract machine, Section 4 defines the instruction set and the interfaces between different execution levels, and the unification algorithm, Section 5 defines the interfaces between Prolog and C code, Section 6 defines the storage format used in file-to-file compilation, Section 7 describes the garbage collector, Section 8 describes the compiler. Section A lists a number of macros used internally by the emulator. Finally, Section B lists the actual bytecode representation used by the emulator.

2 Language and Implementation

2.1 The Language

Industrial SICStus Prolog follows the mainstream Prolog tradition in terms of syntax and built-in predicates, and is largely compatible with DECsystem-10 Prolog and Quintus Prolog [17]. Notably, modules and exception raising and handling and coroutines are part of the language.

The full language is defined in [9].

2.2 The Implementation

The implementation is based on Warren's abstract machine (WAM) [18] with significant extensions. The implementation languages are C and Prolog plus a small set of support routines in assembler for native code predicates. Except for the native code part, the ambition is to make the implementation portable to BSD UNIX machines. The compilation modes are file-to-core and file-to-file. Three levels of execution are incorporated:

Interpreted predicates, supporting program debugging and dynamic database updates, are available by the well-known technique of meta-interpreter. Interpreted clauses are represented as if there were a compiled predicate clause/2, to speed up database retrievals. Notably, interpreted predicates are indexed. The indexing method is described in [10]. Also, the DECsystem-10 Prolog predicates recorda/3, recordz/3, recorded/3 are simulated by using dynamic clauses of prolog:$current:instance/2.

Emulated predicates, simplifying bootstrapping and stressing memory efficiency and fast compilation, are available by compiling Prolog source code to a bytecoded abstract instruction set. The bytecode is interpreted (emulated) by a C program. A subset of the emulator would be needed anyway to perform database searches for the meta-interpreter.

Native Code predicates, stressing fast execution, are available by compiling the intermediate (WAM) code representation to native code.

Thus, the implementation can be regarded as consisting of four parts of comparable sizes: an emulator (in C), a runtime system (in C and assembler), a programming environment (in Prolog), and a compiler (in Prolog).

We have aimed at an implementation with the following properties:
• Portability. By writing all essential parts in C and Prolog and avoiding machine-
specific features, reasonable portability has been achieved.

• Compatibility. The Quintus dialect, recognized as the de facto industrial standard, is
supported as a subset. Some features from modern Prologs such as NU-Prolog [15],
like coroutining primitives, have also been added.

• Several execution levels. We provide three levels of execution: interpreted, supporting
program debugging and dynamic database updates, emulated, simplifying bootstrapping
and stressing memory efficiency and fast compilation, and native code, stressing
fast execution. In addition, foreign language subroutines can be dynamically linked
and called from Prolog.

• Efficiency. This is not an essential property, but increases the general usefulness of the
tool. Efficiency is primarily achieved by an optimizing compiler, emulation techniques,
and by compiling to native code.

3 Storage Model

The abstract machine described herein is a modified Warren abstract machine. Our model
addresses certain issues not treated in the original WAM, e.g. shallow backtracking, arithmetic,
cut, delays, and garbage collection. It inherits all major properties of Warren’s model,
such as structure copying, separate choicepoints and environments, and tagged pointers.

The storage model is similar to the original WAM, except that the local stack has been
split into a choicepoint stack and an environment stack. Unbound variables are represented
as self-references.

3.1 Terms and their Representation

The main types of terms are variables, constants, and compound terms.

The variables in a given Prolog clause are statically classified as permanent or temporary.
A variable is permanent if it occurs both before and after a procedure call, otherwise it is
temporary. Clauses that contain permanent variables will store their bound values in an
environment at run time. In WAM notation, temporary registers are denoted by \( X_n \), and
permanent registers by \( Y_n \). For example, in:

\[ p([X, X_s], Z) : - q(X_s, R_s), r(U, U), s(R_s, [X], Z). \]

\( X, R_s \), and \( Z \) are permanent, and \( U \) and \( X_s \) are temporary.

A variable can be unbound, conditionally bound or unconditionally bound to another
term. A process known as dereferencing follows a chain of bound variables until an unbound
variable or a non-variable is encountered. An unbound variable can be constrained on some
goal or unconstrained. Binding a constrained variable causes the constraint (the goal) to be
invoked. A binding is conditional if there can be another execution path which may bind
the variable to something else. Conditional bindings can be undone and are recorded on the
trail stack (see Section 3.2.5).

Constrained variables are constructed on the heap or global stack (see Section 3.2.2);
unconstrained variables are constructed on the heap (for temporary variables) or on the
environment stack (see Section 3.2.3) (for permanent variables).

A constant is an atom or a number. A compound term is composed of a functor and
up to 255 arguments (arbitrary terms). A list is a special case of compound term with the
functor \( . / 2 \).

Terms are represented as tagged pointers to objects. A tagged pointer has the parts:
where the *tag* distinguishes the type of term, the *value* usually points to an object, and the *GC* bits are reserved for garbage collection. Three bits usually suffice for the *tag*; however, for numbers a fourth bit. The following variants are used:

<table>
<thead>
<tr>
<th>term</th>
<th>bits 28..31</th>
<th>bits 2..27</th>
<th>bits 0..1</th>
</tr>
</thead>
<tbody>
<tr>
<td>heap variable</td>
<td>0000</td>
<td>pointer</td>
<td>GC</td>
</tr>
<tr>
<td>constrained variable</td>
<td>0010</td>
<td>pointer</td>
<td>GC</td>
</tr>
<tr>
<td>stack variable</td>
<td>0100</td>
<td>pointer</td>
<td>GC</td>
</tr>
<tr>
<td>small int</td>
<td>1000</td>
<td>value</td>
<td>GC</td>
</tr>
<tr>
<td>float header</td>
<td>1001</td>
<td>num size</td>
<td>GC</td>
</tr>
<tr>
<td>atom</td>
<td>1010</td>
<td>index</td>
<td>GC</td>
</tr>
<tr>
<td>functor</td>
<td>1010</td>
<td>arity index</td>
<td>GC</td>
</tr>
<tr>
<td>int header</td>
<td>1011</td>
<td>num size</td>
<td>GC</td>
</tr>
<tr>
<td>list</td>
<td>1100</td>
<td>pointer</td>
<td>GC</td>
</tr>
<tr>
<td>structure</td>
<td>1110</td>
<td>pointer</td>
<td>GC</td>
</tr>
</tbody>
</table>

where the various fields mean the following:

**pointer** A pointer to an object on the heap.

For unconstrained variables, the object is just a value cell.

For constrained variables, the object consists of three words \(\langle C, G, D \rangle\) where \(C\) is a value cell. If the variable has a single constraint, \(G\) contains a goal record and \(D\) contains an integer tagged pointer to the struct definition record for the predicate to call. For a variable with more than one constraint, \(G\) and \(D\) are lists, and the head and tail of each list are recursively interpreted as \(\langle G, D \rangle\) pairs. A goal record is either a goal or a list whose head is a mutual exclusion variable and whose tail is a goal. See Section 5.3 for the mechanism operating the mutual exclusion variables.

For lists, the object consists of two words: the head and tail.

For structures, the object consists of \(1 + n\) words where the first word is the functor of arity \(n\) and the remaining words are the arguments.

For floats and big integers, the object consists of \(2 + n\) words where the first and last words contain a header and the remaining words contain the binary representation of the number. The algorithms operating on big integers are essentially Knuth’s [13].

**value** A small integer \(i\) is represented as \(0x88000000 + 4i\).

**index** An 18-bit quantity uniquely defining an atom (index into an internal table).

**arity_index** For a functor \(a/n\), the high 8 bits contain the arity \(n\), and the low 18 bits hold the atom index for the atom \(a\).

**num_size** The value is \(1 + n\), where \(n\) is the number of words required to hold the binary representation of the number (\(n = 2\) for floats).

**GC** This is always zero except during garbage collection.
3.2 Data Areas

The data areas are divided into the static area, for information which is saved from one query to another, and dynamic areas, for information which is not needed upon backtracking. The dynamic areas are operated as stacks; the static area as a memory pool in which objects of arbitrary size can be allocated. The address order of the various areas is not critical, and neither is the growth direction of the stacks.

In contrast to Warren's model, we have split the local stack into an environment stack and a choicepoint stack. There is also a heap, sometimes called a global stack, and a trail stack. There is no explicit PDL, just the implicit C PDL.

A brief description of each memory area follows, listing for each area the kinds of objects that it may contain. The actual record layouts can be found in the source code.

3.2.1 The Static Area

This area is operated by the dynamic memory allocation routines provided by the C library (malloc(), realloc(), and free()). It contains a variety of objects described below.

compiled clauses Clauses of compiled predicates are stored as a linked list of records of type struct emul_info. Any disjunctions in a compiled clause are stored as "internal" predicates under that clause. This holds for bytecode as well as native code clauses.

instances Clauses of an interpreted predicate are stored as a doubly linked list of records of type struct instance. In fact, two sets of forward and backward links are used: one set covers the entire list of clauses, and the other set chains together clauses that have the same indexing key.

The forward links are terminated by NULL; the backward links wrap around. Each instance points back to the beginning of the definition record. A rank field defines a total order on a list.

The semantics for an invocation of a dynamic predicate is that it is unaffected by asserts and retracts until that invocation has finitely failed. To support the proper semantics of calls and updates, two timestamps are associated with each instance. The implementation is based on a scheme described in [14]. The actual implementation differs somewhat from the original scheme. Details are given in comments in the file objareas.c. See also the macro ACTIVE_INSTANCE (Section A) which manages the physical deletion of dead instances.

streams Open streams are stored as a doubly linked list of records of type struct stream_node. The records contain fields for filename, access mode, character and line counters, FILE pointer, and information for socket streams.

atoms Atom indices refer to the array atmtab of pointers to items of the hash table prolog_atoms. Each hash table item points at a record of type struct atom, which itself contains the atom index, thus closing the loop. New atoms are assigned indices in consecutive order.

alternatives For a given goal of an emulated predicate, alternatives that might match the goal are stored as a linked list of records of type struct try_node. The records contain entrypoints into the clause for read and write mode (depending on whether $X_0$ is unbound at runtime), choicepoint size information, and the clause number.

indexing tables Indexing table are used in indexing on first argument in calls and in looking up predicates. They are operated as hash tables with quadratic overflow handling. The are allowed to grow 50% full, after which they are expanded and rehashed.
Hash table access is performed by using some of the low order bits of the key as array index, and then searching for a hit or for a zero key indicating that the key is absent from the table. Records of type struct sw_on_key represent hash tables.

**emulated predicates** An emulated predicate is represented as a record of type incore_info containing a list of clauses, a together with indexing information if the predicate is indexed or just a list of alternatives if it is not indexed.

**predicates** Predicates are stored as records of type struct definition, which contain information about name, arity, exporting and importing, block declarations, native code entry points, and other properties.

There are several basic kinds of predicates: native, undefined, emulated, interpreted, and builtin. In addition, spypoints, can be placed on predicates, and predicates can be subject to dynamic, multifile, and block declarations. The properties field encodes all of this, while the enter_insn field encodes the first action that the emulator should take when the predicate is called.

**statistics** Various information about the runtime and memory consumption of the running program are collected together in a record of type struct statistics. This structure is preserved by the restore/1 operation, which otherwise restores an entire memory image.

### 3.2.2 The Heap

This area is sometimes called the global stack. Each object consists of a sequence of words representing terms. It grows towards increasing addresses during forward execution and contracts on backtracking. It is garbage collected when full.

The following global variables define current heap bounds:

```c
extern TAGGED *heap_start;    /* physical low bound */
extern TAGGED *heap_end;      /* physical high bound */
extern TAGGED *heap_warn;     /* initial heap overflow limit */
extern TAGGED *heap_warn_soft; /* current heap overflow limit */
extern TAGGED *int_heap_warn; /* heap_warn juggled by \tt ctl-C */
```

An unconstrained heap variable may not be bound to a stack variable or to an unconstrained heap variable located at a higher address. A constrained heap variable may not be bound to an unconstrained variable.

### 3.2.3 The Environment Stack

This area contains environments. An environment represents a list of goals still to be executed. It consists of a number of variables which the compiler has classified as permanent occurring in the body of a clause plus a pointer into the body of a continuation clause and its environment. This area grows towards increasing addresses at recursive calls and contracts on determinate calls and on backtracking. It is stack shifted (i.e. expanded with pointer relocation) when full.

The following global variables define current environment stack bounds:

```c
extern TAGGED *stack_start;    /* physical low bound */
extern TAGGED *stack_end;      /* physical high bound */
extern TAGGED *stack_warn;     /* stack overflow limit */
```

An environment is represented by a record
struct frame {
    struct frame *frame; /* environment pointer */
    INSN *next_insn;  /* program pointer */
    TAGGED term[ANY];  /* permanent variables */
};

A permanent variable may be bound to any term, except to another variable on the
environment stack if the other variable is located at a higher address.

At procedure calls, the current environment's active size is found at an offset from the
continuation pointer. This information is needed by certain instructions and is denoted
FrameSize(CP).

3.2.4 The Choicepoint Stack

This area contains choicepoints. A choicepoint is created when entering a procedure Q with
arity n which has more than one clause that could match the goal. When no alternatives
remain, the choicepoint is deleted. It grows towards decreasing addresses i.e. towards the
trail stack which shares the same physical area.

The following global variables define current choicepoint stack bounds:

    extern TAGGED *choice_start; /* physical low bound */
    extern TAGGED *choice_end;  /* physical high bound */
    extern TAGGED *tagged_choice_start; /* low bound as integer */

A choicepoint consists of a snapshot of the crucial abstract machine registers:

struct node {
    TAGGED *trail_top; /* top of trail stack */
    TAGGED *global_top; /* top of global stack */
    struct try_node *next_alt; /* alternative */
    struct frame *frame; /* environment pointer */
    INSN *next_insn;
    struct frame *local_top; /* environment stack pointer */
    TAGGED term[ANY]; /* saved argument registers */
};

3.2.5 The Trail Stack

The main use of this area is to record conditional variable bindings. A variable is condi-
tionally bound iff the variable is older than the youngest choicepoint. Upon backtracking,
entries are simply popped off the trail stack and the bound variables are reset to unbound.

If during backtracking a non-variable is encountered on the trail stack, it is treated as a
goal to be executed upon backtracking. This can be used to ensure that a side-effect that
had effect over a finitely failed subcomputation is undone.

The trail stack grows towards increasing addresses i.e. towards the choicepoint stack
which shares the same physical area. The following global variables define current trail
stack bounds:

    extern TAGGED *trail_start; /* physical low bound */
    extern TAGGED *trail_end;  /* physical high bound */
3.3 Abstract Machine Registers

The current computational state is held in the memory areas and in the abstract machine
registers, pointing into the areas. The registers are collected into a record of type struct
worker, the crucial fields of which are:

```c
struct worker {
    ...
    struct node *node; /* choice pointer */
    struct node *next_node; /* -^-- at predicate entry */
    struct node *segment_node; /* gc's segment choice point */
    INSN insn; /* program counter */
    TAGGED *structure; /* structure pointer */
    TAGGED global_uncond; /* first uncond. global variable */
    TAGGED local_uncond; /* first uncond. local variable */
    struct frame *frame2; /* permanent variable base */
    ...

    /* incidentally, the rest is similar to a struct node */
    TAGGED *trail_top; /* trail pointer */
    TAGGED *global_top; /* global stack pointer */
    struct try_node *next_alt; /* alternative */
    struct frame *frame; /* current environment */
    INSN *next_insn; /* continuation */
    struct frame *local_top; /* environment stack pointer */
    TAGGED term[ANY]; /* temporary variables */
};
```

extern struct worker *self;

In the description of the instructions and builtin predicates, the above structure will be
referred to by the local C variable w as it is usually passed as an argument.

N.B. The local_top, local_var. and segment_node fields are not necessarily kept up
to date by the emulator. When they occur in the description of the instructions and builtin
predicates, they are implicitly updated.

4 The Emulator

The section discusses the interfaces between the various execution levels, the abstract
machine instruction set, and the general unification algorithm.

4.1 Interfaces

An issue of prime importance to the overall design is the coexistence of builtin, interpreted,
emulated, and native code predicates, and the necessary interfacing between them.

Since the emulator is written in C, builtin predicates (also written in C) are similar to
emulated predicates as far as parameter passing and access to abstract machine registers is
concerned.

The interpreted category causes few complications, since meta-interpretation is used,
i.e. the interpreter is a Prolog program. When an interpreted predicate is called, control is
simply transferred to the meta-interpreter.

Native code predicates, on the other hand, need special support to interface well with
the rest of the system. The interface is implemented as a call/return protocol: control is
transferred to native code by a function call and from native code by a function return. In
both directions a message may be passed in:
enum native_value {
    EXECUTE_EMUL,
    PROCEED_EMUL,
    FAIL_EMUL
};
enum native_value native_rc;

There are six possible cases for transfer to or from native code. The emulator code for the various cases is as follows:

1. The emulator calls native code. This is detected since every predicate has a tag defining its type (the `enter_instr` field).

   ```c
   w->insn = Def->code.incoreinfo->othercase->emulcode;
   native_rc = execute_native(Arg);
   goto switch_on_native_value;
   ```

   where Def is the predicate being called.

2. The emulator backtracks into native code. This is detected by introducing a WAM instruction `retry_native` and by using the `emul.p2` field of alternative records (struct `try_node`) as backtracking information for native code. The `emul.p2` field for the `first` alternative in a chain is used in the emulator for different purposes.

   For native predicates, the `emul.p` field points at a `retry_native` instruction immediately followed by the native code while the `emul.p2` field points at a specialized backtracking routine. For emulated predicates, the `emul.p2` field (except for the first alternative) points at a routine `_fail_emul` which transfers control to the emulator’s backtracking routine. See Section 4.2.6.

3. The emulator proceeds into a native continuation, thus completing a native call to an emulated predicate. This action is defined as its own WAM instruction, `leave`. Before reverting to native code, an environment is deallocated. See Section 4.2.6.

4. Native code calls emulated, signalled by returning `EXECUTE_EMUL`. Stack overflow conditions, `ctl-C`, woken delayed goals, and spypoints detected by native code also use this mechanism, thus leaving it up to the emulator to deal with the condition.

5. Native code backtracks into emulated code, signalled by returning `FAIL_EMUL`.

6. Native code proceeds into an emulated continuation, signalled by returning `PROCEED_EMUL`.

Thus the emulator has three ways of invoking native code, and native code has three ways of returning. Upon return to the emulator, the message passed back from native code is inspected:

```c
switch_on_native_value:
    switch (native_rc)
    {
    case EXECUTE_EMUL:
        w->frame2 = w->local_top;       /* allocate */
        w->frame2->frame = w->frame;
        w->frame2->next_insn = w->next_insn;
        w->frame = w->frame2;           /* allocate2 */
        w->next_insn = &{leave};        /* with FrameSize=0 */
    ```
Execute(w->insn);
case FAIL_EMUL:
    Fail;
case PROCEED_EMUL:
    Proceed;
}

The native code interface also consists of several routines in assembler code, including routines corresponding to the parts of the emulator displayed above, except that they are responsible for moving WAM registers to or from native locations.

4.2 Instruction Set

The descriptions of the instructions will need several support macros which are defined in Section A. They attempt to hide implementation details which may obscure the logic of the various instructions. They also hide the underlying memory model from the emulator code.

4.2.1 Put Instructions

These instructions correspond to body arguments. They prepare arguments for the next procedure call.

put_x_void(i)
This represents an ith goal argument that is a singleton variable.

    LoadHVA(w->term[i]);

put_y_void(n)
This represents a permanent variable $Y_n$ that has not yet been initialized at the first call in a clause.

    LoadSVA(_,w->frame2->term[n]);

put_x_variable(n,i)
This represents an ith goal argument that is an uninitialized temporary variable $X_n$ which does not occur after the next procedure call.

    LoadHVA(w->term[n]);
    w->term[i] = w->term[n];

put_y_variable(n,i)
This represents an ith goal argument that is an uninitialized permanent variable $Y_n$ which occurs in the first procedure call and later.

    LoadSVA(w->term[i],w->frame2->term[n]);

put_x_value(n,i)
This represents an ith goal argument that is an initialized temporary variable $X_n$ which cannot point (even before dereferencing) to a portion of the stack which is about to be deallocated.

    w->term[i] = w->term[n];
put_y.value(n,i)
This represents an ith goal argument that is an initialized permanent variable \( Y_n \)
which cannot point (even before dereferencing) to a portion of the stack which is
about to be deallocated.

\[
w->\text{term}[i] = w->\text{frame2}->\text{term}[n];
\]

put_x.unsafe.value(n,i)
This represents an ith goal argument that is an initialized temporary variable \( X_n \)
which might point to a portion of the stack which is about to be deallocated, i.e. it
might need globalizing.

\[
\text{RefStackUnsafe}(w->\text{term}[i],&w->\text{term}[n]);
\]

put_y.unsafe.value(n,i)
This represents an ith goal argument that is an initialized permanent variable \( Y_n \)
which might point to a portion of the stack which is about to be deallocated, i.e. it
might need globalizing.

\[
\text{RefStackUnsafe}(w->\text{term}[i],&w->\text{frame2}->\text{term}[n]);
\]

put_constant(C,i)
This represents an ith goal argument that is the constant \( C \).

\[
w->\text{term}[i] = C;
\]

put_large(C,i)
Same as put_constant(C,i) where \( C \) is the float or big integer \( C \).

put_temp_large(C,i)
Same as put_large(C,i), but informs the native code back-end that the number is
nested inside a larger structure and that there will be exactly one use of \( X_i \) by a
unify_temp_value(i) instruction.

put_nil(i)
This represents an ith goal argument that is the empty list.

\[
w->\text{term}[i] = \text{atom_nil};
\]

put_structure(F,i)
This represents an ith goal argument that is a structure whose functor is \( F \). The
instruction is followed by a sequence of unify instructions.

\[
w->\text{term}[i] = \text{Tag}(\text{STR},w->\text{global_top});
*w->\text{global_top}++ = F;
w->\text{structure} = \text{NULL};
\]

put_temp_structure(F,i)
Same as put_structure(F,i), but informs the native code back-end that the struc-
ture is nested inside a larger structure and that there will be exactly one use of \( X_i \) by
a unify_temp_value(i) instruction.
put_list(i)
    This represents an i\textsuperscript{th} goal argument that is a list. The instruction is followed by a sequence of unify instructions.
    
    \begin{verbatim}
    w->term[i] = Tag(LST, w->global_top);
    w->structure = NULL;
    \end{verbatim}

put_temp_list(i)
    Same as put_list(i), but informs the native code back-end that the list is nested inside a larger structure and that there will be exactly one use of \(X_i\) by a unify_temp_value(i) instruction.

4.2.2 Get Instructions

These instructions correspond to head arguments. They match head arguments against actual arguments.

get_x_variable(n,i)
    This represents an i\textsuperscript{th} head argument that is an uninitialized temporary variable \(X_n\).
    
    \begin{verbatim}
    w->term[n] = w->term[i];
    \end{verbatim}

get_y_variable(n,i)
    This represents an i\textsuperscript{th} head argument that is an uninitialized permanent variable \(Y_n\).
    
    \begin{verbatim}
    w->frame2->term[n] = w->term[i];
    \end{verbatim}

get_y_first_value(n,i)
    This represents an i\textsuperscript{th} head argument that is an uninitialized permanent variable \(Y_n\) occurring after the first procedure call.
    
    \begin{verbatim}
    BindVAR(Tag(SVA, &w->frame2->term[n]), w->term[i]);
    \end{verbatim}

get_x_value(n,i)
    This represents an i\textsuperscript{th} head argument that is an initialized temporary variable \(X_n\).
    
    \begin{verbatim}
    Unify(w->term[n], w->term[i]);
    \end{verbatim}

get_y_value(n,i)
    This represents an i\textsuperscript{th} head argument that is an initialized permanent variable \(Y_n\).
    
    \begin{verbatim}
    Unify(w->frame2->term[n], w->term[i]);
    \end{verbatim}

get_constant(C,i)
    This represents an i\textsuperscript{th} head argument that is the constant \(C\).
    
    \begin{verbatim}
    Unify(C, w->term[i]);
    \end{verbatim}

get_large(C,i)
    Same as get_constant(C,i) where \(C\) is a float or a big integer.
get_temp_large(C,i)
   Same as get_large(C,i), but informs the native code back-end that the number is
   nested inside a larger structure, that $X_i$ was defined by a unify_temp_variable(i)
   instruction and that this is the only use of it.

get_nil(i)
   This represents an $i$th head argument that is the empty list.

   Unify(atom_nil,w->term[i]);

get_structure(F,i)
   This represents an $i$th head argument that is a structure with the functor $F$. The
   instruction is followed by a sequence of unify instructions.

   {
      TAGGED t1;
      t1=w->term[i];
      DerefTerm(t1,{
         BindVAR(t1,Tag(STR,w->global_top));
         *w->global_top++ = F;
         w->structure = NULL;
      },
      {
         if (TagOf(t1)！=STR || *TagToPointer(t1)！=F)
            Fail;
         w->structure = TagToPointer(t1)+1;
      });
   }

get_temp_structure(F,i)
   Same as get_structure(F,i), but informs the native code back-end that
   the structure is nested inside a larger structure, that $X_i$ was defined by a
   unify_temp_variable(i) instruction and that this is the only use of it.

get_list(i)
   This represents an $i$th head argument that is a list. The instruction is followed by a
   sequence of unify instructions.

   {
      TAGGED t1;
      t1=w->term[i];
      DerefTerm(t1,{
         BindVAR(t1,Tag(LST,w->global_top));
         w->structure = NULL;
      },
      {
         if (TagOf(t1)！=LST)
            Fail;
         w->structure = TagToPointer(t1);
      });
   }
get_temp_list(i)
   Same as get_list(i), but informs the native code back-end that the list is nested inside a larger structure, that \( X \); was defined by a \texttt{unify_temp_variable(i)} instruction and that this is the only use of it.

get_constraint(i)
   This represents an \( i \)th head argument that is a constrained variable. This is only supported in dynamic code. The instruction is followed by a sequence of unify instructions which unify the actual constraints.

\begin{verbatim}
   Unify(w->term[i],Tag(CVA,w->global_top++));
   w->structure = NULL;
\end{verbatim}

4.2.3 Unify Instructions

These instructions correspond to arguments of compound terms. Each one has two modes of operation, read and write. In read mode, the structure pointer (\( w->structure \)) points to the next structure argument that the instruction should match. In write mode, the structure pointer is \texttt{NULL} and the next structure argument is written to the top of the heap.

\texttt{unify\_void}
   This represents a structure argument that is a singleton variable.

\begin{verbatim}
   if (w->structure)
      w->structure++;
   else
      ConstrHVA;
\end{verbatim}

\texttt{unify\_x\_variable(n)}
   This represents a structure argument that is an uninitialized temporary variable \( X_n \).

\begin{verbatim}
   if (w->structure)
      w->term[n] = *w->structure++;
   else
      LoadHVA(w->term[n]);
\end{verbatim}

\texttt{unify\_temp\_variable(n)}
   Same as \texttt{unify\_temp\_variable(n)}, but informs the native code back-end that the structure argument is a nested subterm, thus there will be exactly one use of it.

\texttt{unify\_y\_variable(n)}
   This represents a structure argument that is an uninitialized permanent variable \( Y_n \), occurring before the first procedure call.

\begin{verbatim}
   if (w->structure)
      w->frame2->term[n] = *w->structure++;
   else
      LoadHVA(w->frame2->term[n]);
\end{verbatim}

\texttt{unify\_y\_first\_value(n)}
   This represents a structure argument that is an uninitialized permanent variable \( Y_n \), occurring after the first procedure call.
if (w->structure)
{
    TAGGED t1 = *w->structure++;

    BindVAR(Tag(SVA,&w->frame2->term[n]),t1);
}
else
{
    TAGGED t1;

    LoadHVA(t1);
    BindVAR(Tag(SVA,&w->frame2->term[n]),t1);
}

unify_x.value(n)
    This represents a structure argument that is an initialized temporary variable X_n
    which cannot (even before dereferencing) be pointing to the stack.

    if (w->structure)
    {
        TAGGED t1 = *w->structure++;

        Unify(w->term[n],t1);
    }
    else
        *w->global_top++ = w->term[n];

unify_temp.value(n)
    Same as unify_temp.value(n), but informs the native code back-end that X_n holds
    a nested subterm, thus this is the only use of it.

unify_y.value(n)
    This represents a structure argument that is an initialized permanent variable Y_n
    which cannot (even before dereferencing) be pointing to the stack.

    if (w->structure)
    {
        TAGGED t1 = *w->structure++;

        Unify(w->frame2->term[n],t1);
    }
    else
        *w->global_top++ = w->frame2->term[n];

unify_x.local.value(n)
    This represents a structure argument that is an initialized temporary variable X_n
    which could be pointing to the stack, i.e. it could need globalizing.

    if (w->structure)
    {
        TAGGED t1 = *w->structure++;

        Unify(w->term[n],t1);
}  
else  
WriteLocalValue(w->term[n]);

unify_y_local_value(n)  
This represents a structure argument that is an initialized permanent variable Y_n,  
which could be pointing to the stack, i.e. it could need globalizing.

if (w->structure)  
{
    TAGGED t1 = *w->structure++;  
    Unify(w->frame2->term[n], t1);
}  
else  
WriteLocalValue(w->frame2->term[n]);

unify_constant(C)  
This represents a structure argument that is the constant C.

if (w->structure)  
{
    TAGGED t1 = *w->structure++;  
    Unify(C, t1);
}  
else  
*w->global_top++ = C;

unify_large(C)  
Same as unify_constant(C) where C is a float or a big integer.

unify_nil  
This represents a structure argument that is the empty list.

if (w->structure)  
{
    TAGGED t1 = *w->structure++;  
    Unify(atom_nil, t1);
}  
else  
*w->global_top++ = atom_nil;

unify_structure(F)  
This represents a last structure argument that is a structure with the functor F.

if (w->structure)  
{
    TAGGED t1 = *w->structure++;  
    DerefTerm(t1, {  

BindVAR(t1,Tag(STR,w->global_top));
*ww->global_top++ = F;
 ww->structure = NULL;
},
{
 if (TagOf(t1)! = STR || *TagToPointer(t1)!=F)
 Fail;
 ww->structure = TagToPointer(t1)+1;
});

e else
{
 *ww->global_top++ = Tag(STR,ww->global_top+1);
 *ww->global_top++ = F;
}

unify_list
This represents a last structure argument that is a list.

if (ww->structure)
{
 TAGGED t1 = *ww->structure++;

 DerefTerm(t1,{
BindVAR(t1,Tag(LST,ww->global_top));
 ww->structure = NULL;
},
{
 if (TagOf(t1)!=LST)
 Fail;
 ww->structure = TagToPointer(t1);
});

e else
{
 *ww->global_top++ = Tag(LST,ww->global_top+1);
}

4.2.4 Procedural Instructions
These instructions correspond to the head and goals of a clause. They deal with recursive procedure calls and the data structures necessary for them.

allocate
This appears at the beginning of a clause which contains a procedure call followed by other instructions. It must be matched by a deallocate instruction. Space for a new environment is allocated on the stack.

ww->frame2 = ww->local_top;

allocate2
This appears before the first procedure call of a clause, if the call is followed by other instructions. The new environment is completed.
\begin{verbatim}
  w->frame2->frame = w->frame;
  w->frame2->next_insn = w->next_insn;
  w->frame = w->frame2;
\end{verbatim}

deallocate
This appears after computing the arguments of the last goal in a clause that has an allocate instruction. The current environment is deallocated, updating the current environment and continuation WAM registers.

\begin{verbatim}
  w->frame = w->frame2->frame;
  w->next_insn = w->frame2->next_insn;
\end{verbatim}

call(D,k)
This corresponds to a procedure call that does not terminate a clause. The argument \( k \) is the number of variables that are active in the environment. \( D \) is the address of a definition object.

\begin{verbatim}
  w->next_insn = w->insn;
  Execute(D);
\end{verbatim}

execute(D)
This corresponds to a procedure call that terminates a clause. \( D \) is the address of a definition object.

\begin{verbatim}
  Execute(D);
\end{verbatim}

proceed
This terminates a clause not terminated by a procedure call.

\begin{verbatim}
  Proceed;
\end{verbatim}

fail
This backtracks to the latest choicepoint.

\begin{verbatim}
  Fail;
\end{verbatim}

4.2.5 Indexing Instructions
These instructions interact with the clause indexing mechanism which in the emulator is built into the procedure call mechanism. They also deal with choice point handling.

get_constant_{x0}(C)
This represents a first head argument that is the constant \( C \). \( X_0 \) is already dereferenced and is either uninstantiated or instantiated to \( C \).

\begin{verbatim}
  if (IsVar(w->term[0]))
    BindVAR(w->term[0],C)
\end{verbatim}

get_large_{x0}(C)
Same as \texttt{get constant}_{x0}(C) where \( C \) is a float or a big integer.
get.nil.x0
This represents a first head argument that is the empty list. X0 is already dereferenced and is either uninstantiated or instantiated to the empty list.

if (IsVar(w->term[0]))
    BindVAR(w->term[0], atom_nil)

get.structure.x0(F)
This represents a first head argument that is a structure whose functor is F. X0 is already dereferenced and is either uninstantiated or instantiated to a structure whose functor is F. The instruction is followed by a sequence of unify instructions.

if (IsVar(w->term[0]))
{
    BindVAR(w->term[0], Tag(STR, w->structure));
    *w->global_top++ = F;
    w->structure = NULL;
}
else
    w->structure = TagToPointer(w->term[0])+1;

get.list.x0
This represents a first head argument that is a list. X0 is already dereferenced and is either uninstantiated or instantiated to a list. The instruction is followed by a sequence of unify instructions.

if (IsVar(w->term[0]))
{
    BindVAR(w->term[0], Tag(LST, w->global_top));
    w->structure = NULL;
}
else
    w->structure = TagToPointer(w->term[0]);

branch(0)
This occurs at a point in a clause after a neck(0) instruction and indicates a branch from the nondeterministic head code stream to a shared body code stream.

w->insn += 0;

neck(N)
This represents a point in a clause between the head and the first procedure call, where it is appropriate to decide whether a choice point needs to be allocated or updated. The clause belongs to a predicate of N arguments.

if (w->next_alt)
{
    if (w->node->next_alt!=NULL)
    { /* retry */
        w->node->next_alt = w->next_alt;
    }
    else

{ /* try */
  int i;

  w->node->next_alt = w->next_alt;
  w->node->frame = w->frame;
  w->node->nextInsn = w->next_insn;
  w->node->local_top = w->local_top;

  for (i=OffsetToArity(w->node->next_alt->node_offset); i>0;)
    --i, w->node->term[i] = w->term[i];
  w->next_alt = NULL;
}

dynamic_neck_proceed
  This instruction terminates the bytecode of a dynamic clause. X₃ is unified with a
  pointer to the instance, and the neck(₃) and proceed actions are taken.

4.2.6 Utility Instructions
These instructions deal with cut, builtins, arithmetic, native and foreign code.

choice.x(n)
  This represents the presence of a cut operator in a negation, disjunction or implica-
  tion which is the first procedure call of this clause. The cut will reset the current
  choicepoint from the temporary variable Xₙ.

  w->term[n] = PointerToTerm(w->next_node);

choice.y(n)
  This represents the presence of a cut operator after the first procedure call of this
  clause. The cut will reset the current choicepoint from the permanent variable Yₙ.

  w->frame2->term[n] = PointerToTerm(w->next_node);

cut
  This represents a cut operator before the first procedure call not occurring in a nega-
  tion, disjunction, or implication.

  w->node = w->next_node;
  SetShadowregs;

cut.x(n)
  This represents a cut operator before the first procedure call occurring in a nega-
  tion, disjunction, or implication. The cut will reset the current choicepoint from the
  temporary variable Xₙ.

  w->node = TermToPointer(w->term[n]);
  SetShadowregs;

cut.y(n)
  This represents a cut operator after the first procedure call. The cut will reset the
  current choicepoint from the permanent variable Yₙ.
w->node = TermToPointer(w->frame2->term[n]);
SetShadowregs;

function_1(Name,k,i,h,x)
This corresponds to an application of a builtin function \( X_k = \text{Name}(X_i) \). All other argument registers are preserved. If the value is built on the heap, the instruction ensures that at least \( h \) heap words are available afterwards. If a garbage collection is instigated, temporary variables \([0..x]\) are live.

\[ w->\text{term}[k] = \text{Name}(w->\text{term}[i]); \]

function_2(Name,k,i,j,h,x)
This corresponds to an application of a builtin function \( X_k = \text{Name}(X_i, X_j) \). All other argument registers are preserved. If the value is built on the heap, the instruction ensures that at least \( h \) heap words are available afterwards. If a garbage collection is instigated, temporary variables \([0..x]\) are live.

\[ w->\text{term}[k] = \text{Name}(w->\text{term}[i], w->\text{term}[j]); \]

builtin_1(Name,i)
This corresponds to an application of a builtin predicate \( \text{Name}(X_i) \). All other argument registers are preserved.

\[ \text{if} \left( \neg \text{Name}(w->\text{term}[i]) \right) \]
\[ \text{Fail;} \]

builtin_2(Name,i,j)
This corresponds to an application of a builtin predicate \( \text{Name}(X_i, X_j) \). All other argument registers are preserved.

\[ \text{if} \left( \neg \text{Name}(w->\text{term}[i], w->\text{term}[j]) \right) \]
\[ \text{Fail;} \]

builtin_3(Name,i,j,k)
This corresponds to an application of a builtin predicate \( \text{Name}(X_i, X_j, X_k) \). All other argument registers are preserved.

\[ \text{if} \left( \neg \text{Name}(w->\text{term}[i], w->\text{term}[j], w->\text{term}[k]) \right) \]
\[ \text{Fail;} \]

retry_instance
\( X_2 \) is a pointer to a dynamic clause. If there are more alternatives a \textit{retry} operation is done, setting both \( X_2 \) and \( w->\text{node}->\text{term}[2] \) to point at the next alternative, otherwise a \textit{trust} operation is done, popping the choicepoint and adjusting the shadow registers. Finally the current instance is emulated.

continue
This instruction is the continuation of a condition handler which was invoked to take care of a condition that was raised. The interrupted procedure call is restarted.
{
    int i;
    struct definition *Def = TermToPointer(w->frame2->term[0]);
    for (i=0; i<Def->arity; i++) w->term[i] = w->frame2->term[i+1];
    Deallocate;
    Execute(Def);
}

leave
This represents a transition into a native code continuation.

w->frame = w->frame2->frame;
w->next_insn = w->frame2->next_insn;
native_rc = proceed_native(w);
goto switch_on_native_value;

exit_toplevel
The emulator is exited with a return value depending on the reason for exiting. The return value comes from the most recently executed $exit/1$ goal, see Section 5.3.

retry_c(Q)
A backtracking routine for a built-in predicate is called.

    if (w->next_alt) /* disable shallow backtracking */
        w->node->next_alt = w->next_alt, w->next_alt = NULL;
    if (Q(w))
        Proceed;
    else
        Fail;

retry_native(Q)
This represents a transition into a native code alternative. The instruction is immediately followed by native code instructions.

    if (w->next_alt) /* disable shallow backtracking */
        w->node->next_alt = w->next_alt, w->next_alt = NULL;
    native_rc = retry_native(Arg);
    goto switch_on_native_value;

ci_call(i,j)
This calls a foreign language function with arity $i$ and symbol table index $j$.

ci_inarg(i,j)
This sets up input argument number $i$ for a foreign language call according to format specifier $j$.

ci_outarg(i,j)
This sets up output argument number $i$ before a foreign language call according to format specifier $j$.

ci_retval(i,j)
This treats output argument number $i$ after a foreign language call according to format specifier $j$.  

21
heapmargin\_call(i,j)

This represents the fact that a clause may require that at least \( i \) cells remain in the heap. The clause belongs to a predicate with arity \( j \). If less than \( i \) cells remain in the heap, a heap overflow routine is invoked.

### 4.3 General Unification

The general unification algorithm can be expressed as a decision table. Here, \texttt{SVA} denotes a variable on the environment stack, \texttt{HVA} denotes a variable on the heap, and \texttt{CVA} denotes a constrained variable on the heap. \texttt{CONST} and \texttt{COMPLEX} denote constant and compound term, respectively.

The terms to be unified are first dereferenced and compared. If the two tagged pointers are found equal, unification succeeds, otherwise the table is consulted.

<table>
<thead>
<tr>
<th></th>
<th>SVA</th>
<th>HVA</th>
<th>CVA</th>
<th>CONST</th>
<th>COMPLEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVA</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>HVA</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>CVA</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>CONST</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>COMPLEX</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>9</td>
<td>7</td>
</tr>
</tbody>
</table>

The numbers in the table describe the action taken:

1. A stack variable is bound the other term.

\[
\text{BindVAR}(x,y);
\]

2. A heap variable is bound the other term.

\[
\text{BindVAR}(x,y);
\]

3. A constrained variable is bound the other term.

\[
\text{BindVAR}(x,y);
\]

4. Both terms are stack variables. The younger one is bound to the other one.

\[
\text{if } (x > y) \\
\quad \text{BindVAR}(x,y); \\
\text{else} \\
\quad \text{BindVAR}(y,x);
\]

5. Both terms are heap variables. The younger one is bound to the other one.

\[
\text{if } (x > y) \\
\quad \text{BindVAR}(x,y); \\
\text{else} \\
\quad \text{BindVAR}(y,x);
\]

6. Both terms are constrained variables. The younger one is bound to the other one.
if (x > y)
    BindVAR(x,y);
else
    BindVAR(y,x);

7. If the two compound terms have the same functor, the algorithm recurses on their arguments, otherwise it fails.

    {
        TAGGED t0, t1;
        int i;

        if (TagOf(x)==TagOf(y)==LST)
            for (i=0, i<2, i++)
                {
                    t0 = TagToPointer(x)[i];
                    t1 = TagToPointer(y)[i];
                    Unify(t0,t1);
                }
        else if (TagOf(x)==TagOf(y)==STR &&
            *TagToPointer(x) == *TagToPointer(y))
            for (i=1, i<=Arity(x), i++)
                {
                    t0 = TagToPointer(x)[i];
                    t1 = TagToPointer(y)[i];
                    Unify(t0,t1);
                }
        else
            Fail;
    }

8. If the two constants are both floats or big integers, their headers and binary representations must be equal.

    {
        int i;

        if (TagOf(x)==TagOf(y)==STR)
            for (i=0; i < *TagToPointer(x)>>2; i++)
                {
                    if (TagToPointer(x)[i] != TagToPointer(y)[i])
                        Fail;
                }
        else
            Fail;
    }


    Fail;
5 Built-in Predicates and Functions

The main code interfaces of the implementations are:

- inline predicates and functions, invoked by WAM and native instructions,
- predicates written in Prolog, invoked by the emulator, and
- predicates written in C and their support functions, invoked by Prolog code

These interfaces are documented in this section.

5.1 Inline Predicates and Functions

The global C array builtintab holds code pointers to all inline predicates and functions. The relation between functor and array index is unfortunately encoded in three different places: in the emulator code (initial.c), in the compiler front-end (plwam.p4), and in the native code back-ends (n68_clause.p4, nsp_clause.p4). The inline predicates and functions are the following:

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>=/=1</td>
<td>sin/1</td>
</tr>
<tr>
<td>atomic/1</td>
<td>cos/1</td>
</tr>
<tr>
<td>float/1</td>
<td>'SUB1 FUNCTION'/1 tan/1</td>
</tr>
<tr>
<td>integer/1</td>
<td>'ADD1 FUNCTION'/1 cot/1</td>
</tr>
<tr>
<td>nonvar/1</td>
<td>integer/1 sinh/1</td>
</tr>
<tr>
<td>number/1</td>
<td>float/1  cosh/1</td>
</tr>
<tr>
<td>var/1</td>
<td>\ /1 tanh/1</td>
</tr>
<tr>
<td>'IF BUILTIN'/1</td>
<td>+/2 coth/1</td>
</tr>
<tr>
<td>==/2</td>
<td>*/2 asin/1</td>
</tr>
<tr>
<td>&lt;=/2</td>
<td>//2 acos/1</td>
</tr>
<tr>
<td>@&lt;/2</td>
<td>///2 atan/1</td>
</tr>
<tr>
<td>@&gt;/2</td>
<td>mod/2 atan2/2</td>
</tr>
<tr>
<td>@=&lt;/2</td>
<td>#/2 acot/1</td>
</tr>
<tr>
<td>@&gt;=/2</td>
<td>\ /2 acot2/2</td>
</tr>
<tr>
<td>=:/2</td>
<td>\ /2 asinh/1</td>
</tr>
<tr>
<td>==/2</td>
<td>&lt;&lt;/2 acosh/1</td>
</tr>
<tr>
<td>&lt;/2</td>
<td>&gt;&gt;=/2 atanh/1</td>
</tr>
<tr>
<td>&gt;/2</td>
<td>min/2 acoth/1</td>
</tr>
<tr>
<td>=&lt;/2</td>
<td>max/2 sqrt/1</td>
</tr>
<tr>
<td>&gt;=/2</td>
<td>abs/2 log/1</td>
</tr>
<tr>
<td>=../2</td>
<td>msb/2 exp/1</td>
</tr>
<tr>
<td>arg/3</td>
<td>round/1 inf/0</td>
</tr>
<tr>
<td>compare/3</td>
<td>truncate/1 nan/0</td>
</tr>
<tr>
<td>functor/3</td>
<td>floor/1 ceiling/1</td>
</tr>
</tbody>
</table>

where 'IF BUILTIN'/1 was introduced to compile inline the if/3 builtin predicate, and 'ADD1 FUNCTION'/1 and 'SUB1 FUNCTION'/1 were introduced to compile inline expressions $x + 1$ and $x - 1$.
5.2 Predicates Called From C

The predicates herein are all defined in the prolog module.

apply(+G,+D)
Invoked by the emulator at a procedure call, when some constrained variables have
been bound and the debugger is switched off. The associated constraints are woken.
G and D and the format of constrained variables are described in Section 3.1.

control_c_handler
A predicate invoked in response to SIGINT (ctl-C).

debug_apply(+G,+D)
Invoked by the emulator at a procedure call, when some constrained variables have
been bound and the debugger is switched on. The associated constraints are woken.
G and D and the format of constrained variables are described in Section 3.1.

debug_goal(+Goal,+Module)
A predicate invoked by the emulator when it has encountered a spypoint or a call to
call/1 with the debugger switched on.

interpret_compiled_goal(+Goal,+Module,+Def)
A predicate responsible for handling calls from compiled to interpreted goal, when the
debugger is switched on.

interpret_goal(+Goal,+Module,+Def)
A predicate responsible for handling calls from compiled to interpreted goal, when the
debugger is switched off.

reboot(+I)
Invoked by the emulator at abortion and reinitialization. The argument I distinguishes
the reboot reason.

socket_io_handler
A predicate invoked in response to socket interrupts.

$throw_c_ball(+Ball)
A predicate invoked when a C predicate wants to raise an exception.

undefined_goal(+Goal,+Module)
A predicate invoked by the emulator when it has encountered an undefined goal.

undefined_native(+Goal,+Module)
A predicate invoked by the emulator when it has encountered a native code predicate
with no indexing code. This causes the missing code to be compiled and executed.

5.3 Builtin C Predicates

Predicates whose names begin with a dollar sign ($) are internal and are hidden from the
user.

Arguments of predicates are suggested by a mode declaration, to indicate proper usage
of the predicate. A mode declaration has the form name(arg, ..., arg) where each arg
can be one of the forms: +ArgName—this argument should be instantiated in goals for the
predicate. -ArgName—this argument should not be instantiated in goals for the predicate.
?ArgName—this argument may or may not be instantiated in goals for the predicate. If
used differently, the predicate may issue an error message.

Certain predicates cause a change of system state, taking the old state (typically uninstantiated)
and new state (typically instantiated) as arguments, see e.g. prompt. For
instance, a goal \texttt{prompt(U,U)} is legal, causing no change of state, just unifying \texttt{U} with the current state.

Certain predicates were introduced to support the Boolean constraint package or the external storage facility and are documented elsewhere [6, 16].

A \textit{predicate spec} takes one of the forms

\textbf{Name/Arity}
the predicate of a particular name and arity

\texttt{(Spec-ClauseNo-DisjNo)/Arity} the disjunction number \texttt{DisjNo} of arity \texttt{Arity} in clause number \texttt{ClauseNo} of the predicate specified by \texttt{Spec}

\textbf{$\$$abolish(+Head,+Module)}
\texttt{Head} is the most general goal for an existing predicate in the module \texttt{Module}. The predicate is made undefined unless it's a built-in predicate, in which the call fails.

\textbf{$\$$apply(+Head,+Ref,?Mutex)}
\texttt{Head} is the most general goal for an existing predicate, \texttt{Ref} is the integer tagged \textbf{struct definition} record for the predicate. If \texttt{Mutex} is bound, nothing happens, otherwise \texttt{Mutex} is bound to \texttt{[]} and the goal is called. This mechanism supports “disjunctive freezing”.

\textbf{atom_chars(?Atom,?CharList)}
See the User's Manual.

\textbf{$\$$atom_mode(+Atom,-Context)}
Depending on \texttt{Atom}’s printname, \texttt{Context} unified with:

0
if \texttt{Atom} is a sequence of alphanumeric characters (including \texttt{.}), starting with a lower case letter.

4
if \texttt{Atom} is one of \texttt{! ; [} \texttt{].}

2
if \texttt{Atom} is a sequence of \textbf{symbol-char} [9] except the atom \texttt{.} and atoms beginning with /\texttt{.}

1
if \texttt{Atom} is anything else, in which case it needs quoting.

\textbf{$\$$bind.Is(+Socket,-PortOut)}
This “binds” \texttt{Socket} to the Internet port \texttt{PortOut}.

\textbf{$\$$bind.U(+Socket,+Name)}
This “binds” \texttt{Socket} to the socket name \texttt{Name}.

\textbf{$\$$block(+Decl)}
\texttt{Decl} is a block declaration for a predicate in the current module. The declaration is added to the set of block declarations for that predicate. Any existing declarations subsumed by the new one are deleted. If the new declaration is subsumed by some old ones, it is not added.

\textbf{$\$$block.declaration(+Goal,+Module,-Decl)}
\texttt{Goal} is a goal for a predicate in the module \texttt{Module}. \texttt{Decl} is unified with the list of block declarations for that predicate.
$\text{blocked\_goal}(\text{+Goal, +Module})$

Goal is a goal for a predicate in the module Module. This succeeds if the goal is currently blocked according to the predicate's block declarations, in which case the goal is suspended on the relevant variables.

$\text{bootversion}$

Prints the current version number and copyright notice on the standard output stream.

$\text{breaklevel}(\text{?Old, ?New})$

Changes the current command interpreter recursion level from Old to New. Both are integers.

$\text{call\_module}(\text{+Goal, +Module})$

Goal is a goal for a predicate in the module Module. The goal is called.

$\text{character\_count}(\text{+Stream, -Count})$

Count is unified with the number of characters read or written on the stream Stream.

$\text{choice\_usage}(\text{-Usage})$

Usage is unified with a list of two integers giving size of the free and used parts of the choicepoint stack in bytes.

$\text{clause\_number}(\text{+Spec, -Count})$

Count is unified with the current number of clauses of the predicate specified by Spec in the current module.

$\text{clean\_up}(\text{+Chpt})$

Chpt is a choice point, encoded as an integer. If Chpt is the parent of the current choice point, two choice points are popped, otherwise no-op.

$\text{close}(\text{+Stream})$

As close/1 in the User's Manual, except Stream is either a system stream(user_input, user_output, user_error) or a stream specifier (stream(X,Y)). In the latter case, the corresponding struct stream_node is deallocated and choicepoints for current_stream/3 are checked so that none refers to Stream.

$\text{compile\_term}(\text{+List, -Instance})$

Instance is unified with an integer tagged new struct instance whose instruction sequence corresponds to matching a head with two arguments: the head and tail of List. If List contains constrained variables, their constraints are encoded as well. Cyclic structures are not supported.

$\text{compiled\_clause}(\text{+Spec, +Ref, +Mode, +IndexData})$

The clause object Ref which is an integer tagged struct emul_info record is added to the emulated predicate indicated by Spec in the type-in module. The atom Mode (either compactcode or fastcode) indicates what kind of clause. The term IndexData is a structure $f(\text{Type, Key[\text{, Base, Woff, Roff}])}$ which contains various information associated with indexing (for emulated clauses) or instruction offsets (for native code clauses).

$\text{compiling}(\text{?Old, ?New})$

Changes the current compilation mode flag from Old to New. Both are atoms. The available values are installation dependent.

$\text{connect\_1}(\text{+Socket, +Machine, +Port, -Stream})$

The socket Socket is connected to the port Port on the machine Machine and to the Prolog stream Stream.
$\text{connect.U(+Socket, +Path, -Stream)}$

The socket Socket is connected to the socket name Path and to the Prolog stream Stream.

$\text{constraint.list(+Ref,-List)}$

List is unified with the list of all unbound CVAs that are more recent than the heap pointer Ref.

$\text{copy_term(?Term,?CopyOfTerm)}$

See the User's Manual. If Term contains constrained variables, their constraints are copied as well. Cyclic structures are not supported.

$\text{counter.values(+Ptr,+Count,-Values)}$

Ptr is a vector of Count profiling counters. Values is unified with a structure, the arguments of which are the actual counter values.

$\text{current_atom(?Atom)}$

See the User's Manual.

$\text{current.clauses(+Head,-Root,+Module)}$

Head is the most general goal of a dynamic predicate in Module. Root is unified with the integer tagged struct definition * record of the predicate.

$\text{current.host(-Host)}$

See the User's Manual.

$\text{current.instance(?Head,?Body,+Root,-Ref)}$

Through backtracking, Head and Body match a dynamic clause of the predicate whose integer tagged struct definition pointer is Root. The integer tagged struct instance is unified with Ref.

$\text{current.interrupt.stream(+Stream,?Old,?New)}$

Changes the current interrupt handler for the socket stream Stream from Old to New. The available values are off (for no handler) or any other atom, which is taken as the name of a predicate of one argument.

$\text{current.key(+Root,-Data,?Name,?Key)}$

Root is the integer tagged struct definition * record of a dynamic predicate. Data is unified with the list of indexing keys for its clauses. If Name or Key are instantiated, the list returned in Data is restricted to values consistent with Name or Key.

$\text{current.module(?Name)}$

See the User's Manual.

$\text{current.predicate(?Name,+Head,+Module)}$

Through backtracking, Name and Head are unified with existing predicates in the module Module.

$\text{current.stream(?FileName,?Mode,?Stream)}$

See the User's Manual.

$\text{debugger.mode}$

This updates certain C variables to reflect the current debugger setting.

$\text{debugger.state(?Old,?New)}$

Changes the current debugger state from Old to New. Both are represented as structures s(GlobalMode,LocalMode,SkipLevel,Depth,Ancestors). The garbage collector knows about the current debugger state and marks all memory it refers to.
$define.predicate(+Spec,+Mode)
The predicate specified by Spec is added as a new predicate to the type-in module. The atom Mode determines the type of the new predicate.

$defrost(-CVar,+Term)
The constrained variable CVar is "disarmed" by binding it to a new variable. If Term is [], the new variable is unconstrained. If Term is [G|B], the new variable is a constrained variable pointing at the heap object (G, D). See Section 3.1 for details about the representation of constrained variables.

$diff.pivots(?X,?Y,-U,-V)
Fails if the terms X and Y are not unifiable. Otherwise, U and V are unified respectively a variable and a non-variable at the same position in X and Y if such a pair exists. Otherwise, U and V are unified with a pair of non-identical variables at the same position in X and Y if such a pair exists. Otherwise, U and V are unified with [].

$disjunctive.geler(+Vars,+Goal)
The goal Goal in the Prolog module is disjunctively suspended on the variables in the list Vars. If any of the variables is instantiated, however, the call fails.

$display(?Term)

$display(+Stream,?Term)
The principal functor of the term Term is displayed on Stream, which defaults to the current output stream, without quoting.

$empty godef.bin
Physically erases predicate definition and compiled clauses, as these do not get physically erased right away. Only called from the interpreter toplevel.

$emulated.clause.counters(+Spec,+Module,+ClauseNumber,-CounterRef,-Count)
A pointer to the profiling counter vector and its length for the predicate Spec in module Module, clause number ClauseNumber, are unified with CounterRef and Count.

$envstack.usage(-List)
List is unified with a list of two integers giving size of the free and used parts of the environment stack in bytes.

$eq(?X,?Y)
True is X and Y are the same tagged pointer. Not currently in use.

$erase(+Instance)
Instance is an integer tagged struct instance record. It is marked as erased and is, under certain conditions, physically erased.

$erase.clause(+Clause)
Clause is an integer tagged struct emul_info record. It is physically erased. This is used when the user is prompted and refuses to redefine a predicate.

$exit(+Code)
This causes premature exit from the emulator, implementing halt/0 (Code = 0), reinitialise/0 (Code = 1), abort/0 (Code = 2). It is implemented as a forced cut and fail to a choicpoint whose only alternative is an exit_toplevel instruction (see Section 4.2.6).

fail
Instigates backtracking. See the emulator macro Fail.
$fastcode(-Mach)
If native code support is available, Mach is unified with an atom denoting the machine type, otherwise the predicate fails. The currently defined machine types are nc68k and ncsparc.

$ferror_flag(?Old,?New)
Changes the current file error flag from Old to New. Both are on or off.
This flag governs the action taken when a file opening operation fails: If it is on, the computation is aborted, otherwise the operation merely fails.

$find_file(+LibDir,+Path,+Suffix,-Found,-AbsolutePath,-AbsoluteDir)
This predicate attempts to locate an existing file with an optional suffix. LibDir is a library in which to search for path; Path is a path, may be absolute or relative to LibDir; Suffix is an optional suffix to Path; Found is unified with true or fail if a file was found or not. AbsolutePath is unified with the absolute pathname and AbsoluteDir with its directory name.

$first_instance(+Root,-Ref)
Root is the integer tagged struct definition pointing at a list of instances. Ref is unified with the integer tagged struct instance for the first instance, if one exists.

$flush.output(+Stream)
Flushes the buffer of the output stream Stream.

$foreign.base(-Base)
Base is unified with the number of foreign functions loaded so far.

$format.print.float(+Character,+Precision,+Value)

$format.print.integer(+Character,+Precision,+Value)
These predicates serve the "e", "E", "f", "g", "G" "d", "D", "r", and "R" format specs.

$frozen(-Var,-Constraints)
Constraints is unified with the data structure representing goals whose execution is suspended on the uninstantiated variable Var. If nothing is suspended on Var, Constraints is unified with [] Otherwise, Constraints is unified with a list tagged pointer to the second word of its constrained variable record, the details of which are described in Section 3.1.

garbage.collect
See the User's Manual.

$gc.marguerie(?Old,?New)
Changes the current garbage collection margin (in bytes) from Old to New. Both are integers.
Garbage collection is not attempted if the heap is smaller than the margin. The heap is expanded if a garbage collection fails to reclaim that number of bytes.

$gc.mode(?Old,?New)
Changes the current garbage collection mode from Old to New. Both are on or off or all.

$gc.trace(?Old,?New)
Changes the current garbage collection trace level from Old to New. Both are on or off or terse of verbose.